UNIT-IV

Object Oriented Programming

Topic 1: Concept of class, object and instances

- Like other general-purpose programming languages, Python is also an object-oriented language since its beginning.
- It allows us to develop applications using an Object-Oriented approach. In Python we can easily create and use classes and objects.
- An object-oriented paradigm is to design the program using classes and objects.
- The object is related to real-word entities such as book, house, pencil, etc. The oops concept focuses on writing the reusable code.
- It is a widespread technique to solve the problem by creating objects.

Principles of oop:

- Class
- Object
- Method
- Inheritance
- Polymorphism
- Data Abstraction
- Encapsulation

1.Class:

- The class can be defined as a collection of objects. It is a logical entity that has some specific attributes and methods.
- For example: if you have an employee class, then it should contain an attribute and method, i.e. an email id, name, age, salary, etc.

Syntax:

- **class** ClassName:
- <statement-1>
- .
- <statement-N>

Creating classes in Python:

- class Employee:
- id = 10
- name = "Devansh"
- def display (self):
- print(self.id,self.name)
-
- Here, the self is used as a reference variable, which refers to the current class object.
- It is always the first argument in the function definition. However, using self is optional in the function call.

2.Object

- The object is an entity that has state and behavior. It may be any real-world object like the mouse, keyboard, chair, table, pen, etc.
- Everything in Python is an object, and almost everything has attributes and methods.
- All functions have a built-in attribute __doc__, which returns the docstring defined in the function source code.
- Syntax:
- Object name=classname()

Ex:

```
• class car:
•     def __init__(self,modelname, year):
•         self.modelname = modelname
•         self.year = year
•     def display(self):
•         print(self.modelname,self.year)
•
• c1 = car("Toyota", 2016)
  c1.display()
```

Creating an instance of the class:

```
• class Employee:
•     id = 10
•     name = "John"
•     def display (self):
•         print("ID: %d \nName: %s"%(self.id,self.name))
• # Creating a emp instance of Employee class
• emp = Employee()
• emp.display()
```

output:

```
• ID: 10
• Name: John
```

Delete the Object:

- We can delete the properties of the object or object itself by using the del keyword. Consider the following example.

- **class** Employee:
-     id = 10
-     name = "John"
-
-     **def** display(self):
-         **print**("ID: %d \nName: %s" % (self.id, self.name))
-     # Creating a emp instance of Employee class
-
- emp = Employee()
-
- # Deleting the property of object
- **del** emp.id
- # Deleting the object itself
- **del** emp
- emp.display()

3.Method:

- The method is a function that is associated with an object. In Python, a method is not unique to class instances. Any object type can have methods.

4.Inheritance:

- Inheritance is the most important aspect of object-oriented programming, which simulates the real-world concept of inheritance.
- It specifies that the child object acquires all the properties and behaviors of the parent object.
- By using inheritance, we can create a class which uses all the properties and behavior of another class.
- The new class is known as a derived class or child class, and the one whose properties are acquired is known as a base class or parent class.
- It provides the re-usability of the code.

5.Polymorphism:

- Polymorphism contains two words "poly" and "morphs". Poly means many, and morph means shape.
- By polymorphism, we understand that one task can be performed in different ways.
- For example - you have a class animal, and all animals speak.
- But they speak differently. Here, the "speak" behavior is polymorphic in a sense and depends on the animal.
- So, the abstract "animal" concept does not actually "speak", but specific animals (like dogs and cats) have a concrete implementation of the action "speak".

6.Encapsulation:

- Encapsulation is also an essential aspect of object-oriented programming.
- It is used to restrict access to methods and variables.
- In encapsulation, code and data are wrapped together within a single unit from being modified by accident.

7.Data abstraction:

- Data abstraction and encapsulation both are often used as synonyms.
- Both are nearly synonyms because data abstraction is achieved through encapsulation.
- Abstraction is used to hide internal details and show only functionalities.
- Abstracting something means to give names to things so that the name captures the core of what a function or a whole program does.

Topic: 2 Constructors

- A constructor is a special type of method (function) which is used to initialize the instance members of the class.
- In C++ or Java, the constructor has the same name as its class, but it treats constructor differently in Python. It is used to create an object.
- Constructors can be of two types.
- Parameterized Constructor
- Non-parameterized Constructor

- Constructor definition is executed when we create the object of this class. Constructors also verify that there are enough resources for the object to perform any start-up task.

Creating the constructor in python:

- In Python, the method the __init__() simulates the constructor of the class. This method is called when the class is instantiated.
- It accepts the self-keyword as a first argument which allows accessing the attributes or method of the class.

Ex:

```
•   class Employee:
•       def __init__(self, name, id):
•           self.id = id
•           self.name = name
•       def display(self):
•           print("ID: %d \nName: %s" % (self.id, self.name))


•   emp1 = Employee("John", 101)
•   emp2 = Employee("David", 102)
# accessing display() method to print employee 1 information
•   emp1.display()
# accessing display() method to print employee 2 information
•   emp2.display()
```

Output:

```
•   ID: 101
•   Name: John
•   ID: 102
•   Name: David
```

Constructors can be of two types.

1.Python Non-Parameterized Constructor

- The non-parameterized constructor uses when we do not want to manipulate the value or the constructor that has only self as an argument.

Consider the following example.

```
• class Student:
•     # Constructor - non parameterized
•     def __init__(self):
•         print("This is non parametrized constructor")
•     def show(self,name):
•         print("Hello",name)
• student = Student()
• student.show("John")
```

2.Python Parameterized Constructor

```
• class Student:
•     # Constructor - parameterized
•     def __init__(self, name):
•         print("This is parametrized constructor")
•         self.name = name
•     def show(self):
•         print("Hello",self.name)
• student = Student("John")
• student.show()
```

output:

This is parametrized constructor

Hello John

Topic : 3 Destructors in Python

- The users call Destructor for destroying the object.
- In Python, developers might not need destructors as much it is needed in the C++ language.
- This is because Python has a garbage collector whose function is handling memory management automatically.
- In this article, we will discuss how the destructors in Python works and when the users can use them.
- The __del__() function is used as the destructor function in Python.
- The user can call the __del__() function when all the references of the object have been deleted, and it becomes garbage collected.

Syntax:

```
def __del__(self):
# the body of destructor will be written here.
```

Ex:

```
• class Animals:
•
•     # we will initialize the class
•     def __init__(self):
•         print('The class called Animals is CREATED.')
•
•     # now, we will Call the destructor
•     def __del__(self):
•         print('The destructor is called for deleting the Animals.')
•
```

- object = Animals()
- del object

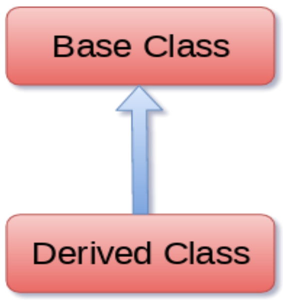
output:

- The class called Animals is CREATED.
- The destructor is called for deleting the Animals.

Topic: 4 Inheritance

- Inheritance is an important aspect of the object-oriented paradigm.
- Inheritance provides code reusability to the program because we can use an existing class to create a new class instead of creating it from scratch.
- In inheritance, the child class acquires the properties and can access all the data members and functions defined in the parent class.
- A child class can also provide its specific implementation to the functions of the parent class. In this section of the tutorial, we will discuss inheritance in detail.

1.Python Single inheritance



Syntax:

- **class** derived-**class**(base **class**):
-     <**class**-suite>

A class can inherit multiple classes by mentioning all of them inside the bracket. Consider the following syntax:

- **class** derive-**class**(<base **class** 1>, <base **class** 2>, ... <base **class** n>):

<**class** - suite>

Ex:

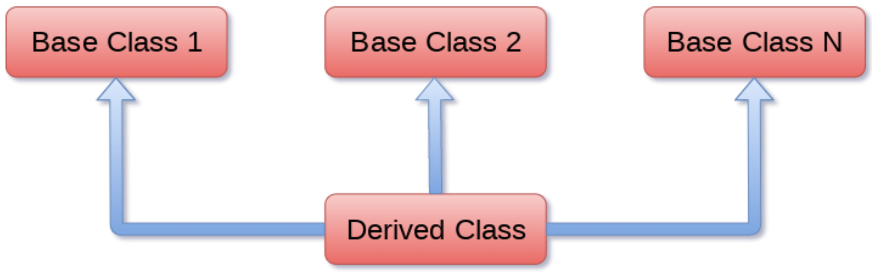
- **class** Animal:
-     **def** speak(self):
-         **print**("Animal Speaking")
- #child class Dog inherits the base class Animal
- **class** Dog(Animal):
-     **def** bark(self):
-         **print**("dog barking")
- d = Dog()
- d.bark()
- d.speak()

Output:

- dog barking
- Animal Speaking
- Eating bread...

2.Python Multiple inheritance:

- Python provides us the flexibility to inherit multiple base classes in the child class.



Syntax:

```
•  class Base1:
•      <class-suite>
•
•  class Base2:
•      <class-suite>
•  .
•  .
•  class BaseN:
•      <class-suite>
•
•  class Derived(Base1, Base2, ...... BaseN):
•      <class-suite>
```

Ex:

```
•  class Calculation1:
•      def Summation(self,a,b):
•          return a+b;
•  class Calculation2:
•      def Multiplication(self,a,b):
•          return a*b;
•  class Derived(Calculation1,Calculation2):
•      def Divide(self,a,b):
•          return a/b;
•  d = Derived()
•  print(d.Summation(10,20))
```

- **print**(d.Multiplication(10,20))
- **print**(d.Divide(10,20))

output:

- 30
- 200
- 0.5

Topic : 5 overlapping and overloading operators

Operator overloading is nothing but same name but differerent type of arguments or return type

Ex: print(1*7)

print(“hii"+”hello”)

Overlapping means :the elements of tuple1 is at least one element equal to tuple2

Then this is called overlapping

Ex: q=(1,2,7)

W=(1,9,7)

method overloading:

- Methods in Python can be called with zero, one, or more parameters.
- This process of calling the same method in different ways is called **method overloading**.
- It is one of the important concepts in OOP. Two methods cannot have the same name in Python;
- hence method overloading is a feature that allows the same operator to have different meanings.

Example:

```
•   class hai:
•       def Hello(self, name=None):
•           if name is not None:
•               print('Hello ' + name)
•           else:
•               print('Hello ')
•
•   # Create an instance
•   obj = hai()
```

- # Call the method
- obj.Hello()
-
- # Call the method with a parameter
- obj.Hello('Kadambini')

Advantages of method overloading in Python:

- reduces complexities
- improves the quality of the code
- is also used for reusability and easy accessibility

Method Overriding:

- We can provide some specific implementation of the parent class method in our child class.
- When the parent class method is defined in the child class with some specific implementation, then the concept is called method overriding.
- We may need to perform method overriding in the scenario where the different definition of a parent class method is needed in the child class.

Ex:

```
class Animal:
    def speak(self):
        print("speaking")
class Dog(Animal):
    def speak(self):
        print("Barking")
d = Dog()
d.speak()
```

Output:

```
Barking
```

Topic: 6 Adding and retrieving dynamic attributes of classes

Dynamic attributes in Python are terminologies for attributes that are defined at runtime, after creating the objects or instances. In Python we call all functions, methods also as an object. So you can define a dynamic instance attribute for nearly anything in Python. Consider the below example for better understanding about the topic.

Example 1:

```
class GFG:

    None

def value():

    return 10

# Driver Code

g = GFG()

# Dynamic attribute of a

# class object

g.d1 = value

# Dynamic attribute of a

# function

value.d1 = "Geeks"

 print(value.d1)

print(g.d1() == value())
```

Output:

```
Geeks

True
```

Design with Classes

Topic : 7 Objects and Classes:

Programmers who use objects and classes know several things:

• The interface or set of methods that can be used with a class of objects

• The attributes of an object that describe its state from the user's point of view

• How to instantiate a class to obtain an object

A class definition is like a blueprint for each of the objects of that class. This blueprint contains

• Definitions of all of the methods that its objects recognize

• Descriptions of the data structures used to maintain the state of an object, or its attributes, from the implementer's point of view.

A First Example: The Student Class

```
>>> from student import Student
>>> s = Student("Maria", 5)
>>> print(s)
Name: Maria
Scores: 0 0 0 0 0
>>> s.setScore(1, 100)
>>> print(s)
Name: Maria
Scores: 100 0 0 0 0
>>> s.getHighScore()
100
>>> s.getAverage()
20
>>> s.getScore(1)
100
>>> s.getName()
'Maria'
```

Topic: 8 Data modeling Examples

As you have seen, objects and classes are useful for modeling objects in the real world. In this section, we explore several other examples.

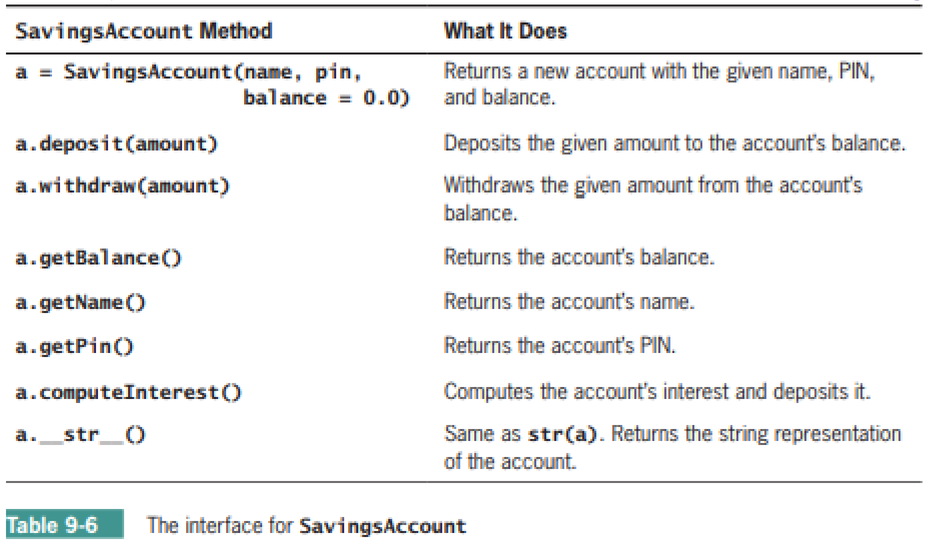
Rational Numbers:

We begin with numbers. A rational number consists of two integer parts, a numerator and a denominator, and is written using the format numerator / denominator. Examples are 1/2, 1/3, and so forth. Operations on rational numbers include arithmetic and comparisons. Python has no built-in type for rational numbers. Let us develop a new class named Rational to support this type of data.

```
>>> oneHalf = Rational(1, 2)
>>> oneSixth = Rational(1, 6)
>>> print(oneHalf)
1/2
>>> print(oneHalf + oneSixth)
2/3
>>> oneHalf == oneSixth
False
>>> oneHalf > oneSixth
True
```

Savings Accounts and Class Variables:

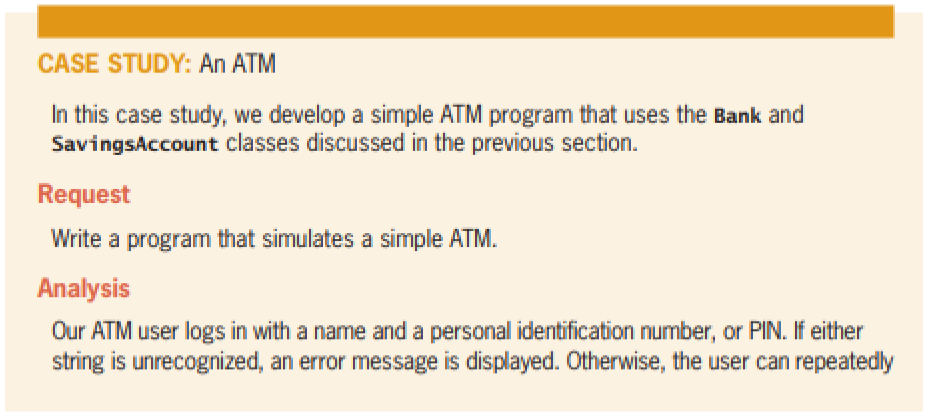
Turning to the world of finance, banking systems are easily modeled with classes. For example, a savings account allows owners to make deposits and withdrawals. These accounts also compute interest periodically. A simplified version of a savings account includes an owner’s name, PIN, and balance as attributes. The interface for a SavingsAccount class is listed in below:



SavingsAccount Method	What It Does
`a = SavingsAccount(name, pin, balance = 0.0)`	Returns a new account with the given name, PIN, and balance.
`a.deposit(amount)`	Deposits the given amount to the account's balance.
`a.withdraw(amount)`	Withdraws the given amount from the account's balance.
`a.getBalance()`	Returns the account's balance.
`a.getName()`	Returns the account's name.
`a.getPin()`	Returns the account's PIN.
`a.computeInterest()`	Computes the account's interest and deposits it.
`a.__str__()`	Same as `str(a)`. Returns the string representation of the account.

Table 9-6 The interface for **SavingsAccount**

Topic:9 Case Study An ATM



CASE STUDY: An ATM

In this case study, we develop a simple ATM program that uses the **Bank** and **SavingsAccount** classes discussed in the previous section.

Request

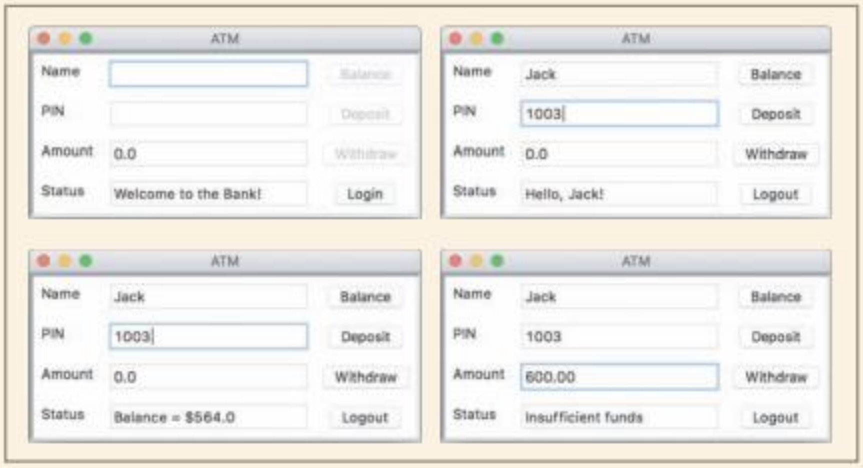
Write a program that simulates a simple ATM.

Analysis

Our ATM user logs in with a name and a personal identification number, or PIN. If either string is unrecognized, an error message is displayed. Otherwise, the user can repeatedly

(continued)

select options to get the balance, make a deposit, and make a withdrawal. A final option allows the user to log out. Figure 9-2 shows the sample interface for this application.



ATM

Name		Balance
PIN		Deposit
Amount	0.0	Withdraw
Status	Welcome to the Bank!	Login

ATM

Name	Jack	Balance
PIN	1003	Deposit
Amount	0.0	Withdraw
Status	Hello, Jack!	Logout

ATM

Name	Jack	Balance
PIN	1003	Deposit
Amount	0.0	Withdraw
Status	Balance = $564.0	Logout

ATM

Name	Jack	Balance
PIN	1003	Deposit
Amount	600.00	Withdraw
Status	Insufficient funds	Logout

Figure 9-2 The user interface for the ATM program

The data model classes for the program are the **Bank** and **SavingsAccount** classes developed earlier in this chapter. To support user interaction, we also develop a new class called **ATM**. The **class diagram** in Figure 9-3 shows the relationships among these classes.

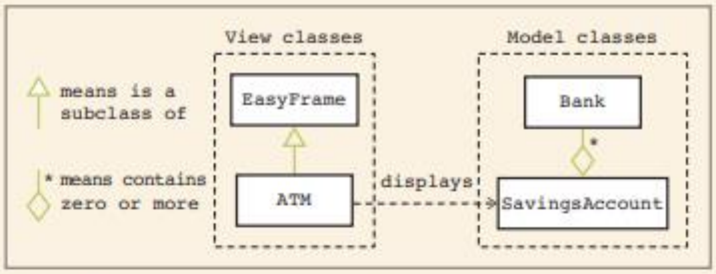
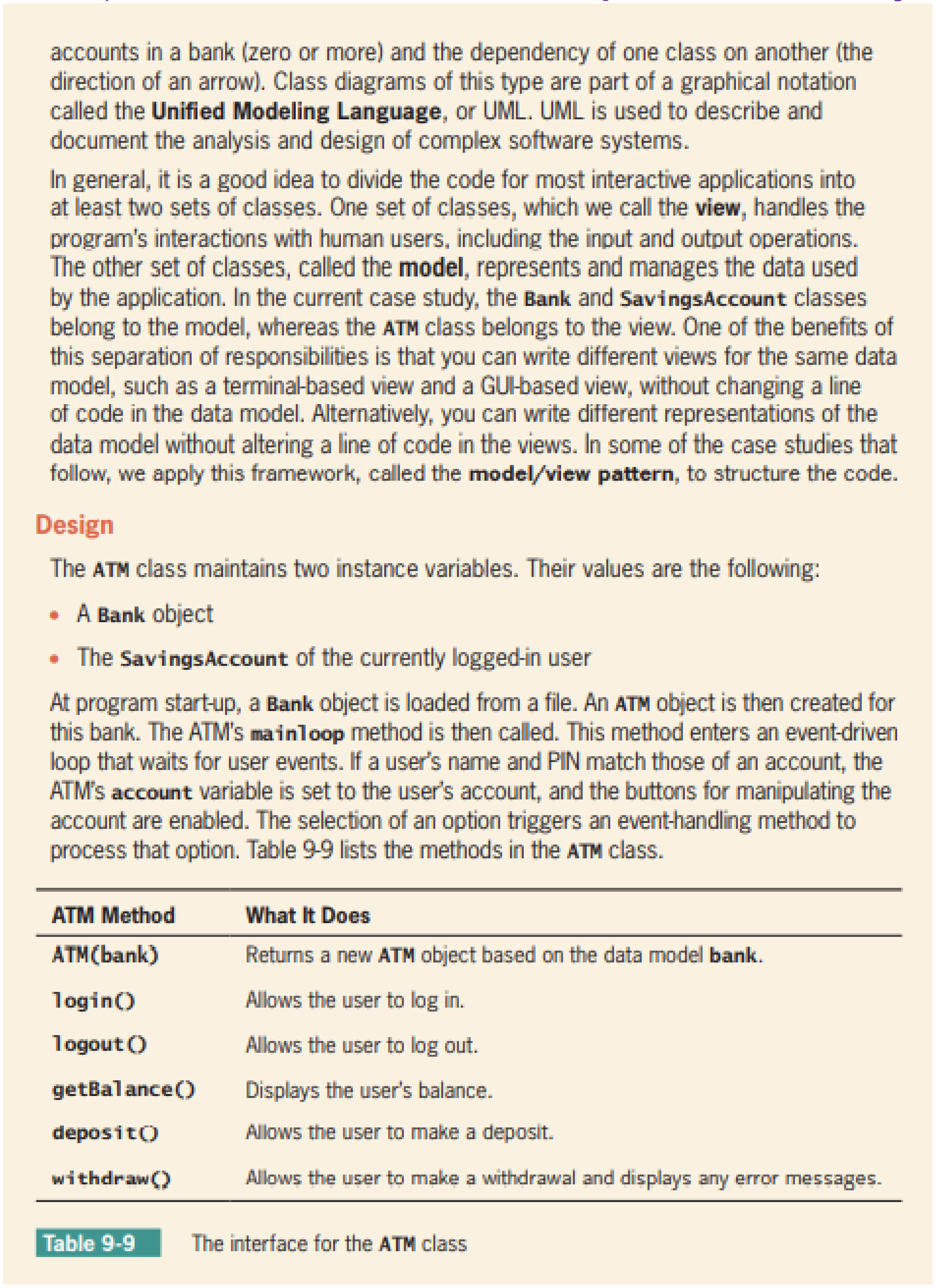


Figure 9-3 A UML diagram for the ATM program showing the program's classes

As you learned in Chapter 8, in a class diagram the name of each class appears in a box. The lines or edges connecting the boxes show the relationships. Note that these edges are labeled or contain arrows. This information describes the number of

(continues)



accounts in a bank (zero or more) and the dependency of one class on another (the direction of an arrow). Class diagrams of this type are part of a graphical notation called the **Unified Modeling Language**, or UML. UML is used to describe and document the analysis and design of complex software systems.

In general, it is a good idea to divide the code for most interactive applications into at least two sets of classes. One set of classes, which we call the **view**, handles the program's interactions with human users, including the input and output operations. The other set of classes, called the **model**, represents and manages the data used by the application. In the current case study, the `Bank` and `SavingsAccount` classes belong to the model, whereas the `ATM` class belongs to the view. One of the benefits of this separation of responsibilities is that you can write different views for the same data model, such as a terminal-based view and a GUI-based view, without changing a line of code in the data model. Alternatively, you can write different representations of the data model without altering a line of code in the views. In some of the case studies that follow, we apply this framework, called the **model/view pattern**, to structure the code.

Design

The `ATM` class maintains two instance variables. Their values are the following:

- A `Bank` object
- The `SavingsAccount` of the currently logged-in user

At program start-up, a `Bank` object is loaded from a file. An `ATM` object is then created for this bank. The ATM's `mainloop` method is then called. This method enters an event-driven loop that waits for user events. If a user's name and PIN match those of an account, the ATM's `account` variable is set to the user's account, and the buttons for manipulating the account are enabled. The selection of an option triggers an event-handling method to process that option. Table 9-9 lists the methods in the `ATM` class.

ATM Method	What It Does
`ATM(bank)`	Returns a new `ATM` object based on the data model `bank`.
`login()`	Allows the user to log in.
`logout()`	Allows the user to log out.
`getBalance()`	Displays the user's balance.
`deposit()`	Allows the user to make a deposit.
`withdraw()`	Allows the user to make a withdrawal and displays any error messages.

Table 9-9 The interface for the `ATM` class

The **ATM** constructor receives a **Bank** object as an argument and saves a reference to it in an instance variable. It also sets its **account** variable to **None**.

Implementation (Coding)

The data model classes **Bank** and **SavingsAccount** are already available in **bank.py** and **savingsaccount.py**. The code for the GUI, in **atm.py**, includes definitions of a main window class named **ATM** and a **main** function. We discuss this function and several of the **ATM** methods, without presenting the complete implementation here.

Before you can run this program, you need to create a bank. For testing purposes, we include in the **Bank** class a simple function named **createBank** that creates and returns a **Bank** object with a number of dummy accounts. Alternatively, the program can load a bank object that has been saved in a file, as discussed earlier.

The **main** function creates a bank and passes this object to the constructor of the **ATM** class. The **ATM** object's **mainloop** method is then run to pop up the window. Here is the code for the imports and the **main** function:

```
"""
File: atm.py
This module defines the ATM class, which provides a window
for bank customers to perform deposits, withdrawals, and
check balances.
"""

from breezypythongui import EasyFrame
from bank import Bank, createBank

# Code for the ATM class goes here (in atm.py)

def main(fileName = None):
    """Creates the bank with the optional file name,
    wraps the window around it, and opens the window.
    Saves the bank when the window closes."""
    if not fileName:
        bank = createBank(5)
    else:
        bank = Bank(fileName)
    print(bank)              # For testing only
    atm = ATM(bank)
    atm.mainloop()
    # Could save the bank to a file here.

if __name__ == "__main__":
     main()
```

(continues)

(continued)

Note that when you launch this as a standalone program, you open the ATM on a bank with 5 dummy accounts; but if you run **main** with a filename argument in the IDLE shell, you open the ATM on a bank created from a saved bank file.

The **__init__** method of **ATM** receives a **Bank** object as an argument and saves a reference to it in an instance variable. This step connects the view (**ATM**) to the model (**Bank**) for the application. The **ATM** object also keeps a reference to the currently open account, which has an initial value of **None**. Here is the code for this method, which omits the straightforward, but rather lengthy and tedious, step of adding the widgets to the window:

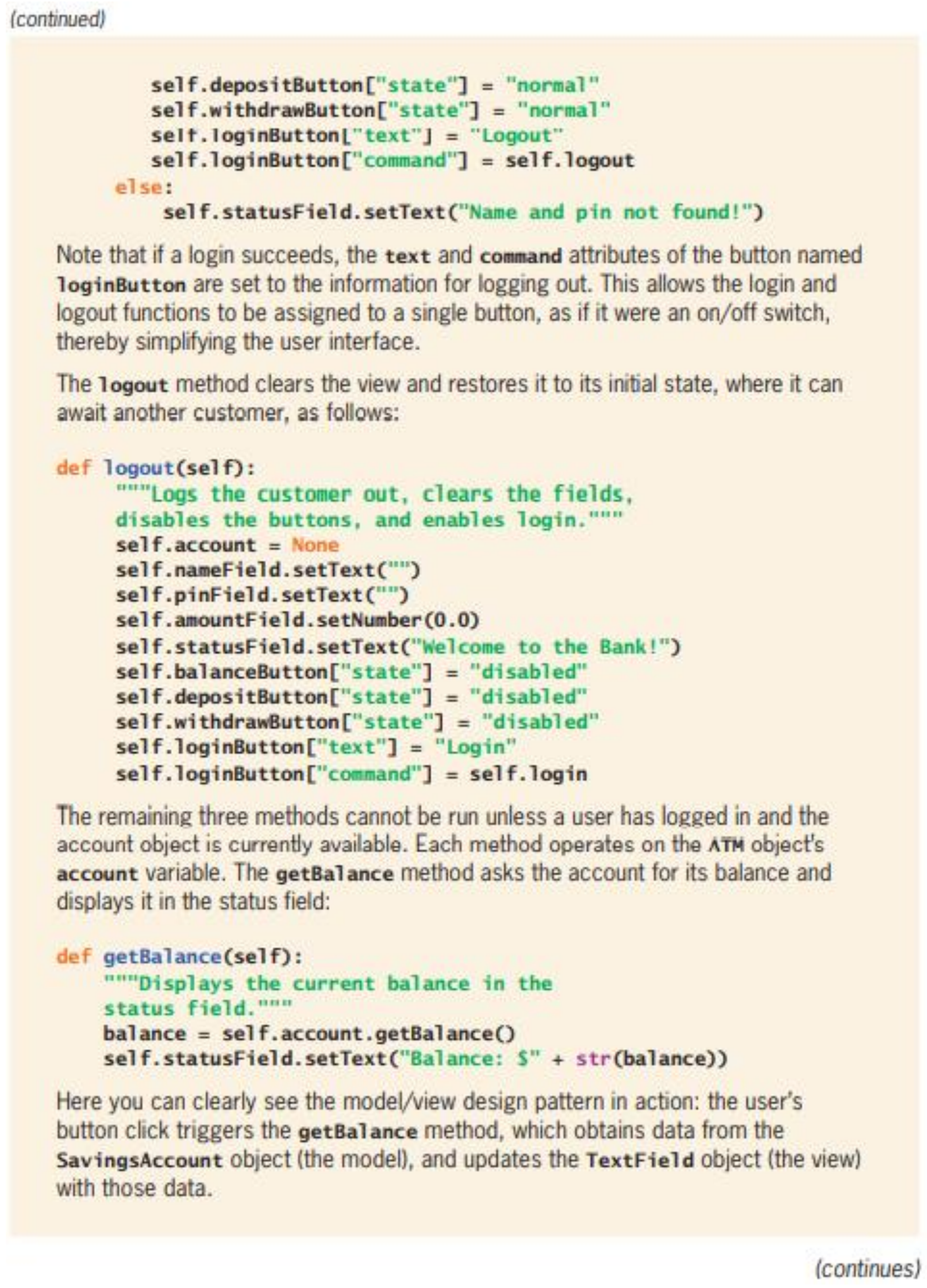
```
class ATM(EasyFrame):
    """Represents an ATM window.
    The window tracks the bank and the current account.
    The current account is None at startup and logout.
    """

    def __init__(self, bank):
        """Initialize the window and establish
        the data model."""
        EasyFrame.__init__(self, title = "ATM")
        # Create references to the data model.
        self.bank = bank
        self.account = None
        # Create and add the widgets to the window.
        # Detailed code available in atm.py

    # Event handling methods go here
```

The event handling method to log the user in takes the username and pin from the input fields and attempts to retrieve an account with these credentials from the bank. If this step is successful, the **account** variable will refer to this account, a greeting will be displayed in the status area, and the buttons to manipulate the account will be enabled. Otherwise, the program displays an error message in the status area. Here is the code for the method **login**:

```
def login(self):
    """Attempts to login the customer.  If successful,
    enables the buttons, including logout."""
    name = self.nameField.getText()
    pin = self.pinField.getText()
    self.account = self.bank.get(name, pin)
    if self.account:
        self.statusField.setText("Hello, " + name + "!")
        self.balanceButton["state"] = "normal"
```



(continued)

```
            self.depositButton["state"] = "normal"
            self.withdrawButton["state"] = "normal"
            self.loginButton["text"] = "Logout"
            self.loginButton["command"] = self.logout
        else:
             self.statusField.setText("Name and pin not found!")
```

Note that if a login succeeds, the **text** and **command** attributes of the button named **loginButton** are set to the information for logging out. This allows the login and logout functions to be assigned to a single button, as if it were an on/off switch, thereby simplifying the user interface.

The **logout** method clears the view and restores it to its initial state, where it can await another customer, as follows:

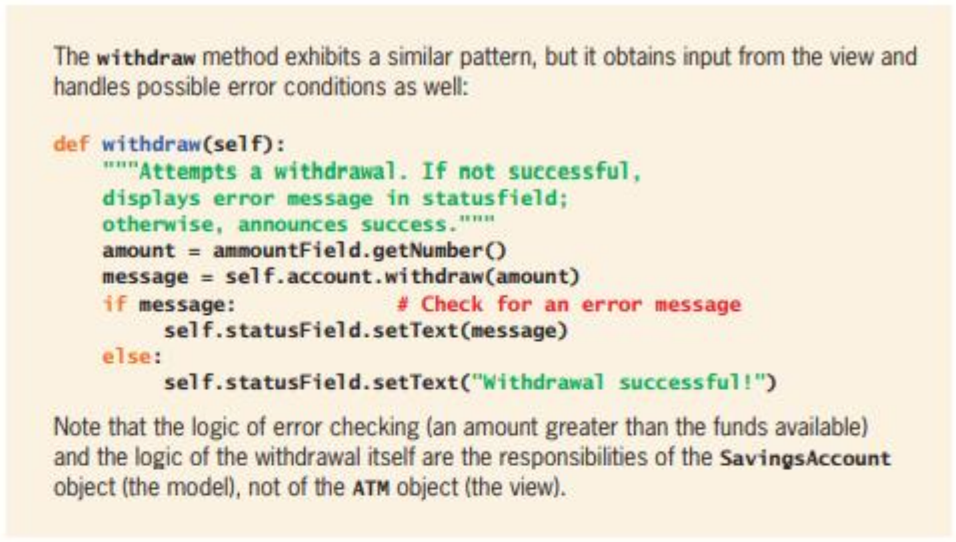
```
def logout(self):
     """Logs the customer out, clears the fields,
     disables the buttons, and enables login."""
     self.account = None
     self.nameField.setText("")
     self.pinField.setText("")
     self.amountField.setNumber(0.0)
     self.statusField.setText("Welcome to the Bank!")
     self.balanceButton["state"] = "disabled"
     self.depositButton["state"] = "disabled"
     self.withdrawButton["state"] = "disabled"
     self.loginButton["text"] = "Login"
     self.loginButton["command"] = self.login
```

The remaining three methods cannot be run unless a user has logged in and the account object is currently available. Each method operates on the **ATM** object's **account** variable. The **getBalance** method asks the account for its balance and displays it in the status field:

```
def getBalance(self):
    """Displays the current balance in the
    status field."""
    balance = self.account.getBalance()
    self.statusField.setText("Balance: $" + str(balance))
```

Here you can clearly see the model/view design pattern in action: the user's button click triggers the **getBalance** method, which obtains data from the **SavingsAccount** object (the model), and updates the **TextField** object (the view) with those data.

(continues)



The **withdraw** method exhibits a similar pattern, but it obtains input from the view and handles possible error conditions as well:

```
def withdraw(self):
    """Attempts a withdrawal. If not successful,
    displays error message in statusfield;
    otherwise, announces success."""
    amount = ammountField.getNumber()
    message = self.account.withdraw(amount)
    if message:                    # Check for an error message
        self.statusField.setText(message)
    else:
        self.statusField.setText("Withdrawal successful!")
```

Note that the logic of error checking (an amount greater than the funds available) and the logic of the withdrawal itself are the responsibilities of the **SavingsAccount** object (the model), not of the **ATM** object (the view).

Topic: 10 Structuring Classes with Inheritance and Polymorphism

Object-based programming involves the use of objects, classes, and methods to solve problems. Object-oriented programming requires the programmer to master the following additional concepts:

1. Data encapsulation. Restricting the manipulation of an object's state by external users to a set of method calls.

2. Inheritance. Allowing a class to automatically reuse and extend the code of similar but more general classes.

3. Polymorphism. Allowing several different classes to use the same general method names.

- Although Python is considered an object-oriented language, its syntax does not enforce data encapsulation. As you have seen, in the case of simple container objects, like playing cards, with little special behavior, it is handy to be able to access the objects' data without a method call.

Inheritance Hierarchies and Modeling:

Objects in the natural world and objects in the world of artifacts can be classified using inheritance hierarchies. A simplified hierarchy of natural objects is depicted in below Figure

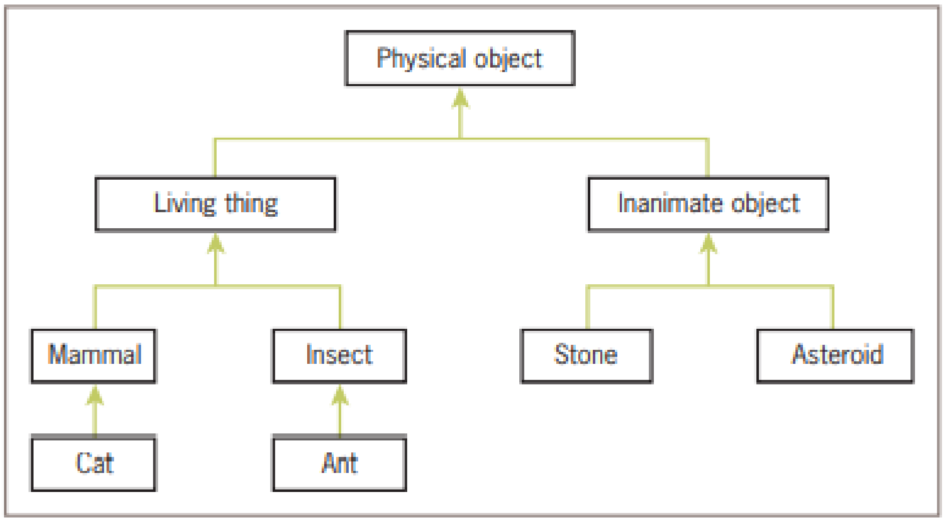


Figure 9-5 A simplified hierarchy of objects in the natural world

Polymorphic Methods:

As we have seen in our two examples, a subclass inherits data and methods from its parent class. We would not bother subclassing unless the two classes shared a substantial amount of abstract behavior.

The Costs and Benefits of Object-Oriented Programming

1. Object-oriented programming attempts to control the complexity of a program while still modeling data that change their state. This style divides up the data into relatively small units called objects.
2. Although object-oriented programming has become quite popular, it can be overused and abused. Many small and medium-sized problems can still be solved effectively
3. To conclude, whatever programming style or combination of styles you choose to solve a problem, good design and common sense are essential

File Operations

The file handling plays an important role when the data needs to be stored permanently into the file. A file is a named location on disk to store related information. We can access the stored information (non-volatile) after the program termination.

The file-handling implementation is slightly lengthy or complicated in the other programming language, but it is easier and shorter in Python.

In Python, files are treated in two modes as text or binary. The file may be in the text or binary format, and each line of a file is ended with the special character.

Hence, a file operation can be done in the following order.

- Open a file
- Read or write - Performing operation
- Close the file

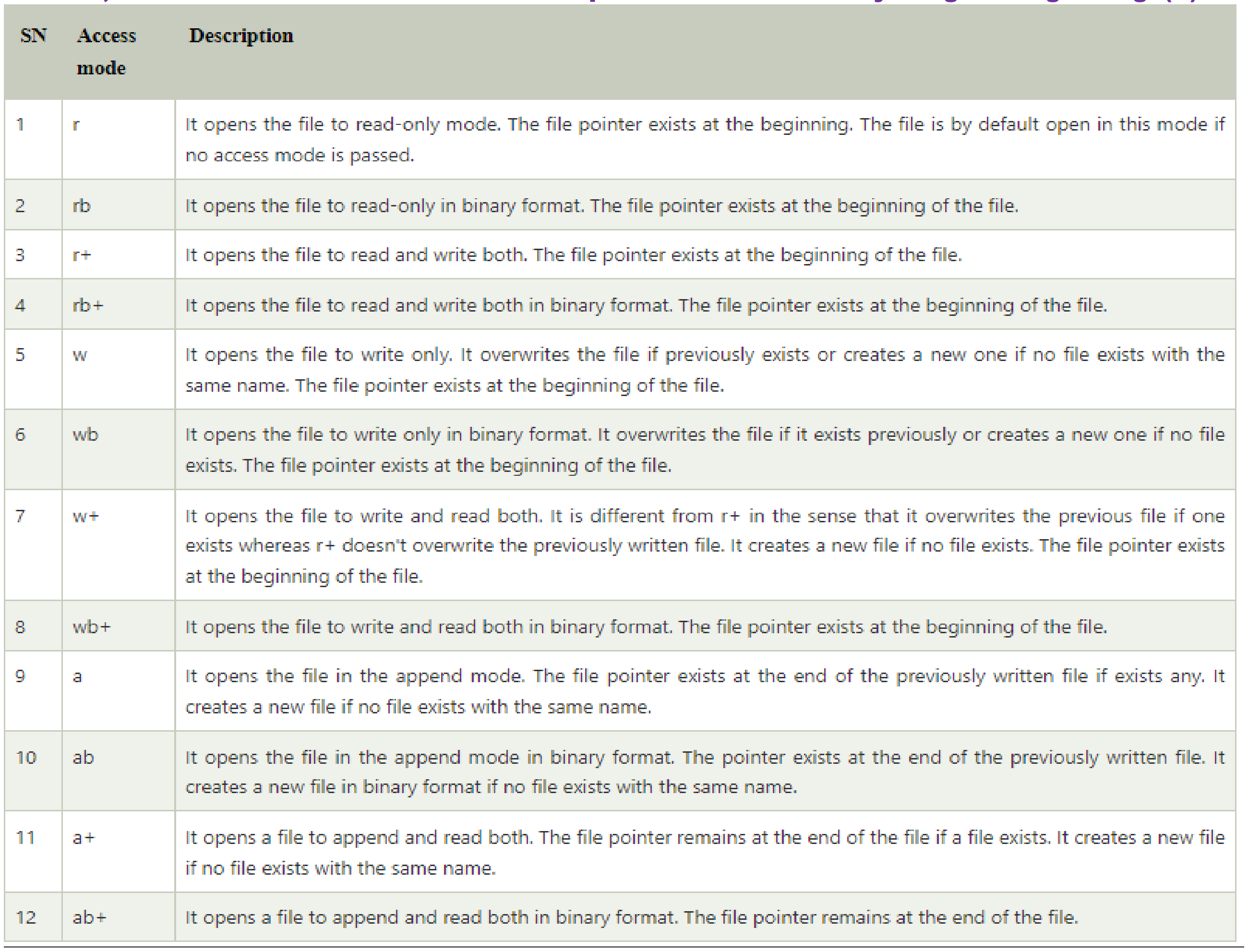
Opening a file

Python provides an **open()** function that accepts two arguments, file name and access mode in which the file is accessed. The function returns a file object which can be used to perform various operations like reading, writing, etc.

Syntax:

1. file object = open(<file-name>, <access-mode>, <buffering>)

The files can be accessed using various modes like read, write, or append. The following are the details about the access mode to open a file.



SN	Access mode	Description
1	r	It opens the file to read-only mode. The file pointer exists at the beginning. The file is by default open in this mode if no access mode is passed.
2	rb	It opens the file to read-only in binary format. The file pointer exists at the beginning of the file.
3	r+	It opens the file to read and write both. The file pointer exists at the beginning of the file.
4	rb+	It opens the file to read and write both in binary format. The file pointer exists at the beginning of the file.
5	w	It opens the file to write only. It overwrites the file if previously exists or creates a new one if no file exists with the same name. The file pointer exists at the beginning of the file.
6	wb	It opens the file to write only in binary format. It overwrites the file if it exists previously or creates a new one if no file exists. The file pointer exists at the beginning of the file.
7	w+	It opens the file to write and read both. It is different from r+ in the sense that it overwrites the previous file if one exists whereas r+ doesn't overwrite the previously written file. It creates a new file if no file exists. The file pointer exists at the beginning of the file.
8	wb+	It opens the file to write and read both in binary format. The file pointer exists at the beginning of the file.
9	a	It opens the file in the append mode. The file pointer exists at the end of the previously written file if exists any. It creates a new file if no file exists with the same name.
10	ab	It opens the file in the append mode in binary format. The pointer exists at the end of the previously written file. It creates a new file in binary format if no file exists with the same name.
11	a+	It opens a file to append and read both. The file pointer remains at the end of the file if a file exists. It creates a new file if no file exists with the same name.
12	ab+	It opens a file to append and read both in binary format. The file pointer remains at the end of the file.

Let's look at the simple example to open a file named "file.txt" (stored in the same directory) in read mode and printing its content on the console.

Example

```
#opens the file file.txt in read mode
fileptr = open("file.txt","r")

if fileptr:
    print("file is opened successfully")
```

Output:

```
<class '_io.TextIOWrapper'>
file is opened successfully
```

The close() method

Once all the operations are done on the file, we must close it through our Python script using the **close()** method. Any unwritten information gets destroyed once the **close()** method is called on a file object.

We can perform any operation on the file externally using the file system which is the currently opened in Python; hence it is good practice to close the file once all the operations are done.

The syntax to use the **close()** method is given below.

Syntax

```
1. fileobject.close()
```

Topic 11: : Reading config files in python, understanding read() function

We can read the file using for loop. Consider the following example.

```
1. #open the file.txt in read mode. causes an error if no such file exists.
2. fileptr = open("file2.txt","r");
3. #running a for loop

4. for i in fileptr:
5.     print(i) # i contains each line of the file
```

Output:

```
Python is the modern day language.

It makes things so simple.

Python has easy syntax and user-friendly interaction.
```

Read Lines of the file

Python facilitates to read the file line by line by using a function **readline()** method. The **readline()** method reads the lines of the file from the beginning, i.e., if we use the readline() method two times, then we can get the first two lines of the file.

Consider the following example which contains a function **readline()** that reads the first line of our file **"file2.txt"** containing three lines. Consider the following example.

Example 1: Reading lines using readline() function

```
#open the file.txt in read mode. causes error if no such file exists.
fileptr = open("file2.txt","r");
#stores all the data of the file into the variable content
content = fileptr.readline()
content1 = fileptr.readline()
#prints the content of the file
print(content)
print(content1)
#closes the opened file
fileptr.close()
```

Output:

```
Python is the modern day language.

  It makes things so simple.
```

We called the **readline()** function two times that's why it read two lines from the file.

Python provides also the **readlines()** method which is used for the reading lines. It returns the list of the lines till the end of **file(EOF)** is reached.

Example 2: Reading Lines Using readlines() function

```
#open the file.txt in read mode. causes error if no such file exists.
fileptr = open("file2.txt","r");

#stores all the data of the file into the variable content
content = fileptr.readlines()
```

```
#prints the content of the file
print(content)

#closes the opened file
fileptr.close()
```

Output:

```
['Python is the modern day language.\n', 'It makes things so simple.\n', 'Python has easy syntax and user-friendly interaction.']
```

Creating a new file

The new file can be created by using one of the following access modes with the function open().

x: it creates a new file with the specified name. It causes an error a file exists with the same name.

a: It creates a new file with the specified name if no such file exists. It appends the content to the file if the file already exists with the specified name.

w: It creates a new file with the specified name if no such file exists. It overwrites the existing file.

Consider the following example.

Example 1

```
#open the file.txt in read mode. causes error if no such file exists.
fileptr = open("file2.txt","x")
print(fileptr)
if fileptr:
    print("File created successfully")
```

Output:

```
<_io.TextIOWrapper name='file2.txt' mode='x' encoding='cp1252'>
File created successfully
```

File Pointer positions

Python provides the tell() method which is used to print the byte number at which the file pointer currently exists. Consider the following example.

```
# open the file file2.txt in read mode
fileptr = open("file2.txt","r")

#initially the filepointer is at 0
print("The filepointer is at byte :",fileptr.tell())

#reading the content of the file
content = fileptr.read();

#after the read operation file pointer modifies. tell() returns the location of the fileptr.

print("After reading, the filepointer is at:",fileptr.tell())
```

Output:

```
The filepointer is at byte : 0
After reading, the filepointer is at: 117
```

Topic: 12 Understanding write functions

Writing the file:

To write some text to a file, we need to open the file using the open method with one of the following access modes.

w: It will overwrite the file if any file exists. The file pointer is at the beginning of the file.

a: It will append the existing file. The file pointer is at the end of the file. It creates a new file if no file exists.

Consider the following example.

Example

```
# open the file.txt in append mode. Create a new file if no such file exists.
fileptr = open("file2.txt", "w")
```

3. # appending the content to the file
4. fileptr.write("""Python is the modern day language. It makes things so simple.
5. It is the fastest-growing programing language""")
6.
7. # closing the opened the file
8. fileptr.close()

Output:

File2.txt

Python is the modern-day language. It makes things so simple. It is the fastest growing programming language.

Topic:13 Manipulating file pointer using seek

Python file method **seek()** sets the file's current position at the offset. The whence argument is optional and defaults to 0, which means absolute file positioning, other values are 1 which means seek relative to the current position and 2 means seek relative to the file's end.

There is no return value. Note that if the file is opened for appending using either 'a' or 'a+', any seek() operations will be undone at the next write.

Syntax

Following is the syntax for **seek()** method −

fileObject.seek(offset[, whence])

Parameters

- **offset** − This is the position of the read/write pointer within the file.
- **whence** − This is optional and defaults to 0 which means absolute file positioning, other values are 1 which means seek relative to the current position and 2 means seek relative to the file's end.

Example

The following example shows the usage of seek() method.

Python is a great language
Python is a great language
#!/usr/bin/python

```
# Open a file
fo = open("foo.txt", "rw+")
print "Name of the file: ", fo.name

# Assuming file has following 5 lines
# This is 1st line
# This is 2nd line
# This is 3rd line
# This is 4th line
# This is 5th line

line = fo.readline()
print "Read Line: %s" % (line)

# Again set the pointer to the beginning
fo.seek(0, 0)
line = fo.readline()
print "Read Line: %s" % (line)

# Close opend file
fo.close()
```

When we run above program, it produces following result −

```
Name of the file: foo.txt
Read Line: Python is a great language.

Read Line: Python is a great language.
```